ストリームデータ処理の分散並列化実行における マージ処理コスト削減方式

勝沼 聡† 今木 常之† 西澤 格† 藤原 真二†

†株式会社日立製作所 中央研究所 〒185-8601 東京都国分寺市東恋ヶ窪一丁目 280 番地

E-mail: †{satoshi.katsunuma.hb, tsuneyuki.imaki.nn, itaru.nishizawa.cw, shinji.fujiwara.yc}@hitachi.com

あらまし 時々刻々と到着するデータをリアルタイムに処理するストリームデータ処理技術が注目されている．ストリームデータ処理では，大量データを処理するために分散並列化が必要であるが，単純にデータを分割して並列に処理するデータ並列方式では，並列処理結果を時刻順に整列し単一のストリームにマージする処理が性能劣化の要因となる．そこで本論文では，マージ処理において時刻順整列処理を削減する P-SORT を提案する．提案方式を実装し評価した結果，性能劣化を回避でき，16 CPU コア利用時に単一 CPU コア実行比で 13 倍の性能向上を確認した．

キーワード リアルタイム，CEP，ストリーム，分散，並列

Distributed and Parallel Stream Data Processing for Reducing Merge Operation Overhead

Satoshi KATSUNUMA† Tsuneyuki IMAKI† Itaru NISHIZAWA† and Shinji FUJIWARA†

†Central Research Lab., Hitachi Ltd., 1-280, Higashi-koigakubo, Kokubunji-shi, Tokyo, 185-8601, Japan

E-mail: †{satoshi.katsunuma.hb, tsuneyuki.imaki.nn, itaru.nishizawa.cw, shinji.fujiwara.yc}@hitachi.com

Abstract Stream Data Processing has been widely accepted to process time series data in real-time fashion. Distributed and parallel stream data processing is required to process large volumes of data. However, the merge operation in traditional parallel data processing techniques degrades system performance. In this paper, we propose a distributed and parallel stream data processing method called P-SORT to reduce merge operation overhead. We implement P-SORT and evaluate its performance. Experimental result shows that the performance of P-SORT with 16 CPU-core execution is 13 times faster than that of single core execution.

Keyword Real Time, CEP, Stream, Distributed, Parallel

1. はじめに

株自動取引，電子マネー，車両位置，携帯操作，センサデータなど，時々刻々と到着するデータのリアルタイム処理を必要とするアプリケーションが増加している．そしてこのようなアプリケーションを効率的に処理する技術である CEP (Complex Event Processing)，及び CEP を実現するミドルウェアであるストリームデータ処理が注目されている．ストリームデータ処理は Stanford 大，MIT などにより研究され[9][11]，製品化も活発化している[6][7][8].

ストリームデータ処理では，センサデータ，株価情報などの実世界から時々刻々と到来する時系列データであるストリームデータを入力する．そして処理対象のストリームデータを切り取り，メモリ上のウィンドウに保持し，新たなデータが到着する度に保持したデータを更新し出力することで高速処理を実現する．またストリームデータ処理では，データ処理の定義に CQL (Continuous Query Language) [10]等の宣言的なクエリ言語を用いる．CQL は，データベース管理システムで用いられる標準言語 SQL に似た言語であり，複雑なデータ処理を簡潔に記述することが可能である．

センサデータや株取引データなどのリアルタイム処理対象のデータ量は急激な増加傾向にあり，ストリームデータ処理を適用しても単一の計算機では十分な性能が得らないケースが現れてきている．ストリームデータ処理では処理件数に比例して CPU 使用量が増加するため，大量データを処理するにはマルチコア環境[1]やマルチノード環境[2]による分散並列化が必須となる．

ストリームデータ処理の分散並列化は，処理内容を直列に分割して実行する方式(パイプライン並列方式) [1][2][3]と，処理対象データを分割して実行する方式(データ並列方式) [4][5]に分類することができ，それぞ

[1] 単一計算機に複数 CPU コア(以下，単にコアと書く)を搭載した環境

[2] 複数の計算機から構成される環境

れ利害得失がある．実世界を対象とした時系列データ処理を考えた場合，例えばセンサごとに閾値を超えたデータ数を独立して集計するなど，対象データを分割できることが多い．このような処理では，データ並列方式では各データを単一コアに割当てて処理可能であるのに対し，パイプライン並列方式では複数コアでデータを送受信し処理する必要がある．このため通信量はパイプライン並列方式ではコア数に比例して増加し，性能劣化に繋がる．一方，データ並列方式では通信量はコア数に依存せず一定であり，性能への影響は小さい．

しかしながら，従来のデータ並列方式では処理データを単一のストリームにマージする際に，時刻順に全データを整列する処理が発生する．マージ処理ではデータ量に比例した処理が必要なため，本整列処理は性能上のボトルネックとなる．そこで本論文では，マージ処理による性能劣化を軽減する方式として P-SORT (Partial SOrt method in Real-Time merge operation) を提案する．P-SORT では，時刻順整列の範囲を絞ることでマージ処理コストを削減する．

本論文の構成は以下の通りである．まず 2 節でストリームデータ処理のデータ並列方式について述べる．そして，次に 3 節において本論文で提案する P-SORT を説明し，4 節でその評価について述べる．5 節で関連研究をまとめ，最後に 6 節で今後の課題を述べる．

2. ストリームデータ処理のデータ並列方式

2.1. ストリームデータ処理の時刻順出力

以下では，発電所で各機器に設置したセンサによって取得されるデータに，ストリームデータ処理を適用して異常を検出する処理を説明する．

図 1 に CQL で記述されたクエリ及びストリームを示す．「センサデータ」ストリームは「センサ」,「測定値」カラム，及びタイムスタンプ「測定時刻」から構成される．クエリ q1 ではセンサデータストリームを入力とし，センサごとの最近一分間の合計データ数(分データ数)を算出する．またクエリ q2 では，同じくセンサデータストリームを入力とし，センサごとの最近一分間の測定値が閾値 α を超えたデータ数(異常データ数)を算出する．そしてクエリ q3 では，クエリ q1 及び q2 で算出した分データ数と異常データ数から，異常率を算出する．

図 2 に，図 1 で定義したストリーム，クエリの処理動作を示す．まず，発電所の機器に取り付けた各センサのデータをセンサデータストリームとして入力し，クエリ q1～q3 に従ってストリームデータ処理エンジンで逐次処理する．そしてこれらのクエリによりセンサの値が異常か否か判定した処理結果を，時刻順に整列されたデータとして出力する．これにより図 2 に示すように，処理結果を受信するアプリケーションでは，あるセンサにおいて異常が検出されると同時に，各センサの異常値の前後関係を解析することが可能となる．障害原因分析アプリケーションでは，あらかじめデータベースに登録された障害原因パターンとマッチングを取り，障害の原因を分析する．ストリームデータ処理では時刻順の逆転を許容する実行方式[12]も研究されているが，正確な処理結果を必要とするアプリケーションでは，本例で示したように時刻順に整列されたデータが対象となる．

```
ストリーム センサデータ
 (センサ String, 測定値 double,
  タイムスタンプ:測定時刻 Timestamp);

クエリ q1
  SELECT センサ,CNT(*) AS 分データ数
    FROM センサデータ [range 1 minute]
         GROUP BY センサ;
クエリ q2
  SELECT センサ,CNT(*) AS 異常データ数
    FROM センサデータ [range 1 minute]
    WHERE 測定値 > 閾値α
         GROUP BY センサ;
クエリ q3
istream(
  SELECT センサ,
         q2.異常データ数 / q1.分データ数 AS 異常率
    FROM q1, q2
    WHERE q1.センサ = q2.センサ);
```

図 1：クエリ及びストリームの記述例

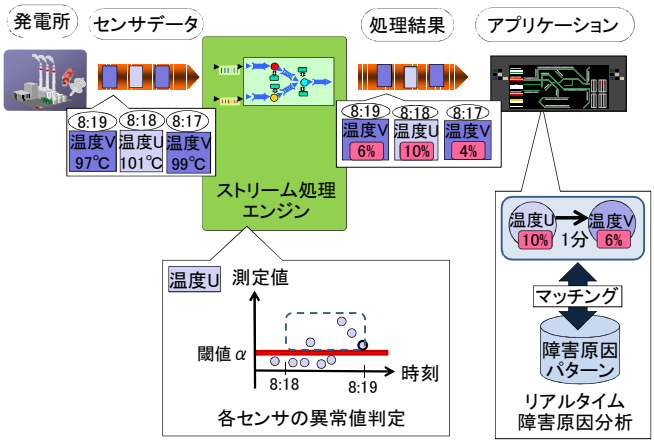

図 2：ストリームデータ処理の動作例

2.2. データ並列方式の動作と効果

ストリームデータ処理では，クエリの処理対象データが，ストリームの特定カラムの値ごとに処理が独立していることが多い．例えば図 1 のクエリ q1～q3 は，センサデータストリームにおけるセンサカラムの値ごとに処理が独立している．このような場合，データ並

列方式では，入力されるストリームデータを特定カラム値に従って複数の CPU コアに振り分ける．そして，各コアで同一クエリの処理を実行し，その処理結果をマージする．

データ並列方式でのデータの割当て方法を説明する．本論文においては，クエリ処理対象のデータが特定のカラムで分割可能であること，及びこのカラムがユーザによって指定されることを仮定する．指定されたカラムは，その値ごとにクエリを独立して処理可能であることを示し，このカラムを用いてデータの振分けテーブルを導出する．振分けテーブルはデータのカラム値をキーとしてコア名を参照するテーブルである．例えば図 1 のクエリ q1～q3 は 2.1 節で述べたように，センサごとに異常値か否かを判定する処理である．この場合，センサカラムの値ごとに処理が独立しているため，クエリ作成者がこのカラムを指定することで，図 3 のようにセンサをキーとする振分けテーブルが導出される．

クエリ実行時には，図 3 に示すように導出した振分けテーブルに従ってデータを複数の CPU コア(本例の場合 CPU コア#1～#4)に振り分け，各コアでクエリ q1～q3 に従ってセンサの異常値判定処理を実行する．そして，処理結果をマージする処理では，データの整合性を確保するため，各 CPU コアで処理したデータを時刻順に整列してアプリケーションに出力する．

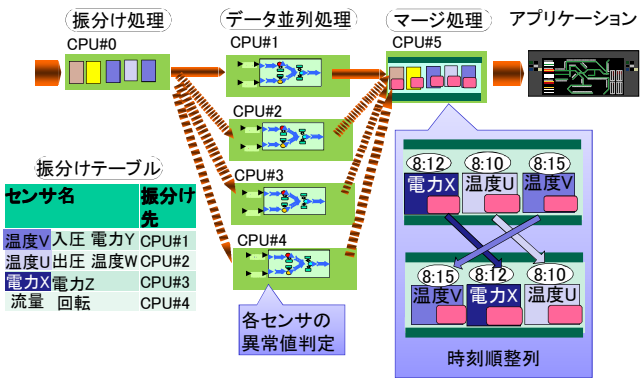

図 3：ストリームデータ処理のデータ並列方式

データ並列方式の効果としては，(1)各コアで処理するデータ数や送受信するデータ数が減ることで CPU 使用量が減少する，(2)コア毎の処理データ数が減ることによりクエリ処理で必要となるメモリ量が減少する，という点が挙げられる．しかし従来のデータ並列方式ではマージ処理が性能上のボトルネックとなる可能性があるため，2.3 節で詳細に検討する．

2.3. データ並列方式におけるマージ処理

データ並列方式におけるマージ処理は，単一コア及び複数コアを用いる二つの方式に大別される．以下ではこれらの方式の動作を説明する．

まず単一コアによるマージ処理では，複数コアで処理されたデータをマージ処理するコア上のキューに転送し，該コアにおいて各キューのデータを時刻順に整列して出力する．例えば図 4 では CPU コア#5 において，CPU コア#1～#4 のデータを対応するキューに格納し，時刻順に整列した後に出力する．

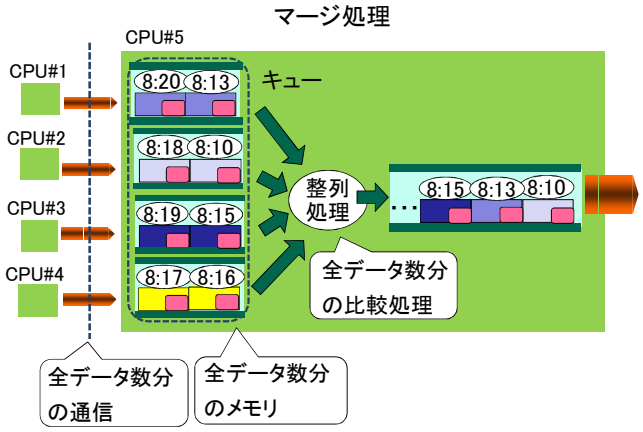

図 4：単一コアによるマージ処理

次に，複数コアによるマージ処理では，まず複数コアで部分的に整列して別コアに転送する．そして転送先のコアで部分的に整列されたデータを再整列することで全データを時刻順に整列する．例えば図 5 では CPU コア#1，#2 において処理されたデータを CPU コア#5 で，CPU コア#3，#4 において処理されたデータを CPU コア#6 で整列し，それらの結果を CPU コア#7 で再整列することで，全データを時刻順に整列する．

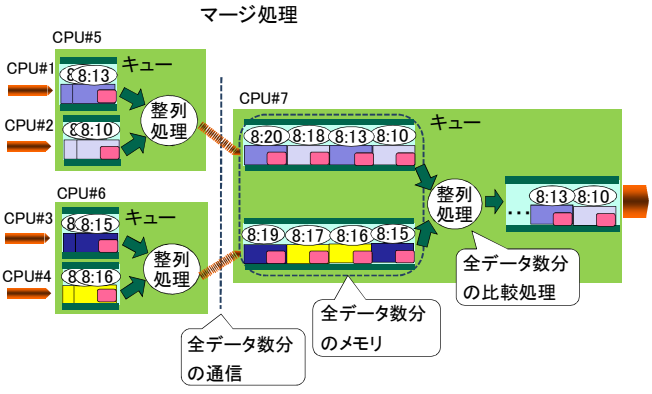

図 5：複数コアによるマージ処理

単一コア及び複数コアによる方式のいずれにおいても，最終的には一つのコアにおいて全データ数分の整列処理が必要になる．例えば単一コアによるマージ処理では，図 4 に示すように CPU コア#5 の 4 個のキューのそれぞれが全データの平均 1/4 ずつのデータ

を受信し，各キューのデータを整列するため，全データ数分の処理が必要になる．また複数コアによるマージ処理でも，図 5 に示すように CPU コア#7 の 2 個のキューが全データの平均 1/2 のデータを受信し，各キューのデータを整列するため，同様に全データ数分の処理が必要になる．このようにマージ処理では単一コアで全データを送受信し，時刻を比較するために，全データ数分の CPU 処理量が必要になる．また，全データを単一コアのキューに格納するため，全データ数分のメモリ量が必要になる．このためマージ処理が性能上のボトルネックとなり，コア数を増やしても一定以上性能が向上しない．

3. マージ処理コスト削減方式の提案

3.1. 時刻順整列範囲の限定

2 節で述べたように，ストリームデータ処理では時刻順にデータを処理する必要があるが，一般の業務において全データに対する時刻順整列が必要な場合は少ない．例えば図 3 の障害原因を分析するアプリケーションでは，一般的に発電所内の特定の障害が関連するセンサは同じ機器，ある機器のグループなどの一定の範囲内に絞られることが多く，その範囲内のセンサを対象として障害を分析する．したがって，各センサデータの時刻順もその範囲内で守られていれば十分である．そこで本論文では，マージ処理による性能の劣化を軽減するために，出力データの時刻順整列を限定する P-SORT を提案する．以下では，P-SORT として，基本 P-SORT, 最適化 P-SORT の二つの方式を説明する．

3.2. 基本 P-SORT:時刻順整列要求による割当て

3.2.1. 基本 P-SORT の方針

データを時刻順に整列する範囲は，同じ処理でも結果を利用するアプリケーションによって異なるため，基本 P-SORT では時刻順整列の要求をユーザが指定可能なインタフェースを提供する．そして時刻順整列要求に従って，整列が必要なデータを同じコアに割当て処理することで整列処理を削減する．

例えば前述のアプリケーションでは，時刻順整列要求を「同一機器のセンサ間で整列が必要」と指定することができる．そして図 6(b)に示すように，同じ機器のセンサデータを単一コアに割り当てる．このことで図 6(a)の従来方式に示されるような全データに対する整列処理を削減できる．

3.2.2. 時刻順整列要求の指定

基本 P-SORT では，ユーザがデータ処理分割キー(Operator Partition Key, 以下 OPK と略す)を指定する．OPK はストリームデータのカラムであり, OPK で指定したカラムの値が異なるデータは，互いに独立して処理可能であることを示す. 例えば, 図 1 のクエリは 2.2 節で述べたようにセンサカラムごとに独立して処理可能であるので，図 7 に示すように OPK としてセンサを指定する．

次に，基本 P-SORT が時刻順整列範囲を算出するために，ユーザは時刻順整列分割キー(Sorting Partition Key，以下 SPK と略す)を指定する．SPK で指定されたカラムで値が同じデータは，時刻順を整列する必要があることを示す．最後に，OPK と SPK の対応関係を示す表として，ユーザは属性対応表を指定する．例えば障害原因分析アプリケーションでは「同一機器のセンサにおいて時刻順整列が必要」であるため，図 7 に示すように SPK として機器が指定される．そして属性対応表として, SPK 及び OPK として指定された機器及びセンサの対応関係が指定される．

3.2.3. データ割当てと整列処理の省略

次にユーザから指定された OPK，SPK，属性対応表を用いて振分けテーブルを導出する．振分けテーブルは，2.2 節で述べたようにデータの振分け時に参照するテーブルであり，振り分けテーブルを参照すること

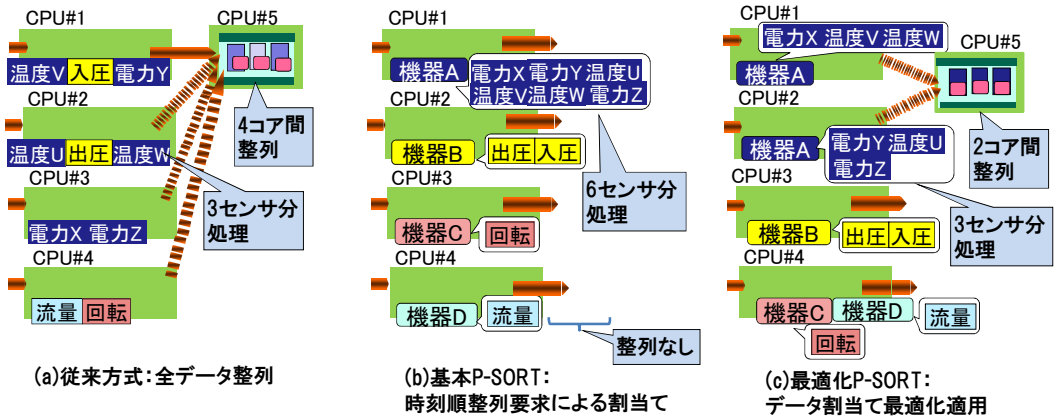

図 6：マージ処理コスト削減に向けたアプローチ

でデータのカラム値をキーとしてそのデータを処理する CPU コア名を取得することができる．基本 P-SORT では，SPK が同じデータは同一の CPU コアに割当てる．そして，属性対応表を参照し，各 CPU コアに割り当てた各 SPK に対応する OPK を抽出し，抽出した OPK の値を振分けテーブルのキーとする．例えば，図 7 では，SPK が機器であるため，機器のとりうる値である{機器 A，機器 B，機器 C，機器 D}を CPU コア#1～#4 に割当てる．また OPK がセンサであるため，属性対応表から機器に対応するセンサを求め，センサをキーとする振分けテーブルを生成する．

そして実行時には，まず振分け処理において，データ振分けテーブルに従って各 CPU コアに入力データを送信する．例えば，図 7 では CPU コア#1～#4 に入力データを振り分ける．そして各 CPU コアで処理した結果は，別 CPU コアの処理結果と時刻順整列することなく，その処理結果を利用する他のストリームデータ処理や，アプリケーションに渡す．なお，この処理結果を利用するストリームデータ処理についても同様に複数コアで実行することができる．

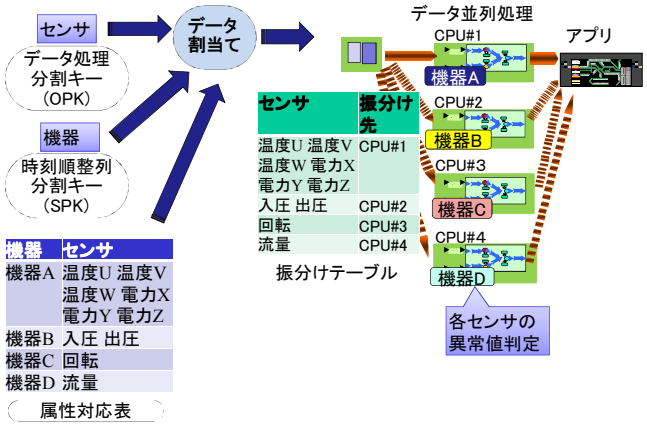

図 7：基本 P-SORT の動作

3.3. 最適化 P-SORT:データ割当て最適化適用

3.3.1. 最適化 P-SORT の方針

基本 P-SORT では，処理内容によっては各コアに処理データを均等に振分けられず，性能が向上しないケースが想定される．そこで最適化 P-SORT では，時刻順整列要求に従ったデータ振分けでコア間の処理データ数に不均衡が生じる場合には，複数コアにデータを割当てる．そして処理後に時刻順整列要求に従って整列処理を行う．データ割り当て時には，コアの処理データ数の最大値を一定以内に抑えつつ，マージ処理によるボトルネックの影響を最小化するようにデータを割り当てる．例えば図 6(b)ではコア#1 の処理データ数が 6 センサ分であり，他のコア#2～#4 よりも多いため，図 6(c)に示すように機器が機器 A であるデータをセンサごとにコア#1，#2 に振分け，処理後コア#5 で整列する．これにより，各コアの処理データ数を最大 3 センサ分に抑えつつ，整列処理も高々2 コア間に留めることができる．

3.3.2. データ割当ての最適化

最適化 P-SORT のデータ割当てには，基本 P-SORT と同様に OPK，SPK，及び属性対応表を用いる．そして，前述したようにコアの処理データ数を一定以内に抑えるために，各コアに割り当てる最大 OPK 数を見積もり，一つの SPK に対応する OPK の数が最大 OPK 数を超える場合には複数コアに割当て，処理後に時刻順整列処理を実行する．また，マージ処理によるボトルネックの影響を最小化するために，対応する OPK の数が多い SPK から優先して割り当てる．

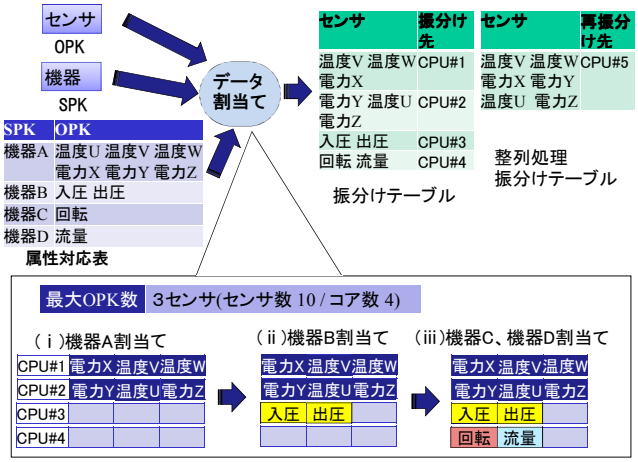

図 8：最適化 P-SORT におけるデータ割当て処理

データ割当ての手順は以下のようになる．

① OPK 数をコア数で割ることにより最大 OPK 数を導出する．

② 各 SPK において，対応する OPK 数を導出する．

③ 対応する OPK 数が多い SPK から順に各コアに割当てる．対応する OPK 数が最大 OPK 数よりも大きい場合には，OPK ごとに複数コアに分けて割当てる．

例えば図 8 に示す例では，センサ数が 10，コア数が 4 であることから，センサ数をコア数で割り，値を切り上げることで最大 OPK 数を 3 と求める．そして(i)機器 A の OPK 数が 6 と最も多いため，最初に機器 A のセンサを複数コア(#1, #2)に割当てる．次に，(ii)OPK 数が二番目に多い機器 B のセンサを単一コア(#3)に割当てる．さらに(iii)機器 C，及び機器 D のセンサをコア#4 に割当てる．

そして，このように決定したデータの割当て方法に従って，基本 P-SORT と同様に振分けテーブルを生成

する．また最適化 P-SORT では整列処理振分けテーブルを生成する．整列処理振分けテーブルは，整列処理するコアにデータを振り分けるために用いるテーブルであり，OPK をキーとして整列処理するコア名を参照する．例えば図 8 では，前述のように機器 A が割り当てられたコア#1，#2 のデータを整列する．したがってコア#1，#2 で処理されるセンサをキーとして整列処理をするコア#5 を参照する整列処理振分けテーブルを生成する．

3.3.3. 部分的な時刻順整列処理

最適化 P-SORT では，従来のデータ並列方式と同様に振分けテーブルに従って各コアに入力データを振り分ける．そして処理されたデータを，整列処理振分けテーブルに従って再振分けし，部分的な時刻順整列を実行する．図 9 は，図 8 で生成した振分けテーブル及び，整列処理振分けテーブルを用いた部分的な時刻順整列処理の動作を示す．図 9 では，振分けテーブルに従ってコア#1，#2 に割り当てられたデータを，整列処理振分けテーブルに従ってコア#5 に送信する．そしてコア#5 で，#1，#2 で処理されたデータ間で時刻順整列し，アプリケーションに出力する．コア#3，#4 に振分けられたデータは整列処理されることなくアプリケーションに出力される．

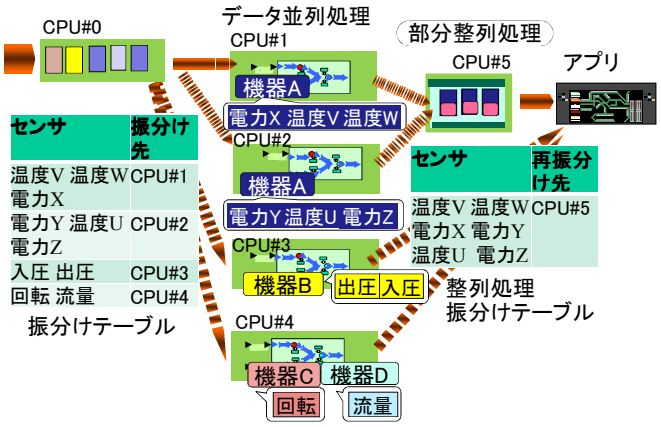

図 9：最適化 P-SORT の部分的な時刻順整列処理

3.3.4. OPK のデータ数に偏りがある場合

各 OPK の処理データ数に偏りがある場合には，データ割当ての最適化において，コアの処理データ数やマージ処理コストを OPK 数で見積もることができない．そこで，OPK ごとのデータ数の分布情報を，OPK，SPK，属性対応表と共にユーザに指定させる．そして，データ数の分布情報を用いて，以下のようにデータを割当てる．

① 最大処理データ数を 1/(コア数)とする．

② 全体の処理データ数に，各 SPK の処理データ数が占める割合を導出する．

③ 処理データ数の占める割合が大きい SPK から順に割当てる．SPK の処理データ数が最大処理データ数よりも大きい場合には，OPK ごとに複数コアに割当てる．

そして，このように決定したデータ割当て方法に従って，3.3.2 節で述べたように，振分けテーブル及び整列処理振分けテーブルを生成し，これらのテーブルを用いて部分的な時刻順整列処理を実行する．

なお各 OPK の処理データ数の比が実行時に変化する場合には，OPK ごとのデータ数の分布情報を実行時に取得し，その情報に従って割当て方法を変更する方法が考えられる．実行時の変更方法の検討については今後の課題とする．

4. マージ処理コスト削減方式の評価

4.1. 評価方法及び環境

ストリームデータ処理のデータ並列方式として，全データを時刻順に整列する従来方式及び，基本 P-SORT，及び最適化 P-SORT をプロトタイプ実装した．そしてプロトタイプを 1～4 台の計算機(4 コア搭載 CPU: Intel® Core™2 Quad，メモリ 4GB)上で動作させて処理性能を測定した．各マシンには OS として Fedora Core Linux 8，Java 仮想マシンとして Sun JVM v1.5 を用いた．Java 仮想マシンには 1 コア当たり 768MB のメモリ領域を割り当てた．

評価に用いたデータは発電機器に取り付けられた各センサからの測定値 1 日分 700 万件である．評価に用いたクエリではこれらのセンサデータを入力し，センサ毎に設定された異常値判定条件に従って，異常値か否か判定する．クエリの処理は，センサごとに独立しているため，従来方式ではセンサごとに複数コアに振分けて処理した．また基本 P-SORT，最適化 P-SORT では OPK をセンサとし，結果を取得するアプリケーションでは機器ごとに分析すると仮定し，SPK を機器とした．

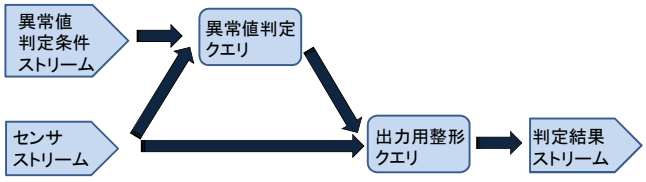

図 10：評価に用いたクエリ

4.2. 評価結果と考察

4.1 節で述べた環境で計算機数，コア数を変化させ，それぞれの実行環境で 700 万件の処理時間を測定し，その処理時間から処理件数(件/秒)を求めた．図 11 は各コア数で実行した場合の，単一コア実行と比べた処

理件数比を表す.

従来のデータ並列方式では，スループット比が 12 コアで 6.0 倍，16 コアで 6.1 倍の性能向上に留まり，ほぼ 16 コア迄に性能向上が止まることを確認した.また，従来方式と同様にセンサごとにデータを振分け，時刻順に整列せずに出力した場合には，12 コアで 11 倍，16 コアで 14 倍と，16 コアにおいても性能向上を確認した．これにより，2 節で述べた時刻順整列処理による性能のボトルネックが確認できた.

一方，基本 P-SORT では 8 コア使用時で 6.1 倍と性能が向上し，従来のデータ並列の 8 コアでの 3.9 倍に対し，整列処理を省いたことによる性能向上を確認した．しかしながら，8 コア以上で実行させた場合には処理性能は向上せず，16 コアで 6.5 倍のスループット向上に留まった．これは，入力データにおいて単一の機器に属するセンサ数が全体のセンサ数の多くを占め，その機器のセンサを割り当てたコアが性能上のボトルネックとなるためである．このように基本 P-SORT では，時刻順整列が必要なデータセットのデータ数の偏りが大きい場合に，性能が頭打ちになってしまうことが確認できた.

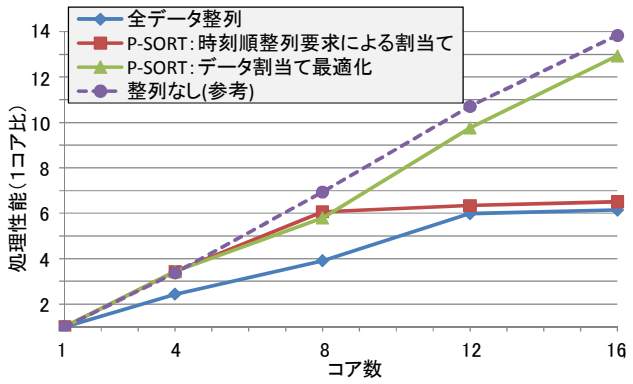

図 11：分散並列化方式の性能評価結果

一方，最適化 P-SORT では 12 コアで 9.8 倍，16 コアで 13 倍と，16 コアにおいても処理性能の向上を確認した．これにより最適化 P-SORT で，コアの最大処理データ数を小さくしつつ，整列処理のボトルネックを最小化する効果が確認できた.実際に 16 コア実行においては，表 1 に示すように，コアの最大処理データ数が全処理データ数の 6.9%となり，従来方式や基本 P-SORT よりも小さいことを確認した．最大処理データ数が従来方式よりも小さくなるのは，クエリの処理をするコア数が従来方式では 16 コア中 12 コアであるのに対し，最適化 P-SORT はマージ処理コストを削減することにより，16 コア中 15 コアに増加したためである．また整列対象のデータ数も全処理データ数の 17%に留まった.

表 1：コアの処理データ数とマージ処理コスト比較

	従来方式	基本 P-SORT	最適化 P-SORT
コア当たりの最大処理データ数(全データ数比)	8.5%	17%	6.9%
整列対象データ数(全データ数比)	100%	0%	17%

なお本評価では，全てのセンサが等間隔でデータを発生すると仮定した．しかし，実際にはデータの種類によって偏る場合がある．このような場合には，3.3.4 節で述べたデータ数の分布情報を用いたデータの割当てが必要になる．このような場合の評価は今後の課題とする.

また，本論文ではデータ処理のレイテンシについては評価していない.時刻順整列するデータ数が増加し，データを処理するコア数が増加すると各コアの処理時間差が大きくなるためレイテンシが大きくなる．したがって従来方式は，P-SORT と比較してレイテンシも増大すると考えられる．レイテンシの評価についても今後の課題とする.

5. 関連研究

ストリームデータ処理の分散並列化の研究は，パイプライン並列方式及びデータ並列方式に大別できる.パイプライン並列方式としては，各クエリのコアへの割当て方法を実行時に変更する方式[1]や，入力ストリームが異なるクエリを各コアに最適に振り分ける方式[2][3]がある．しかしパイプライン並列方式では，2 節で述べたように複数コアでデータを送受信し処理することから，通信による性能劣化が大きい.

一方，データ並列方式としては，動的にデータを再振り分けする方式[5]が検討されている．しかしながら，[5]ではデータ振分け後のマージ処理において，時刻順整列の範囲を狭めることを考慮しておらず，マージ処理のオーバヘッドを削減できないと推測する.

データ並列方式において，処理分割キーを自動的に導出する方式[4]がある.[4]では単一クエリの処理分割キーを抽出し，その処理分割キーから複数クエリの処理分割キーを導出するが，P-SORT とは異なり各コアにデータを均等に振り分けられない場合を考慮していない.そのため,コア間の処理データ数の偏りにより，性能向上しないケースが想定される．また，[4]はマージ処理において時刻順整列の範囲を狭めることを考慮していないため，マージ処理によるオーバヘッドが大

きいと推測する．

6. おわりに

本論文では，ストリームデータ処理の分散並列化において，マージ処理によるボトルネックを回避するP-SORTを提案した．P-SORTでは時刻順整列の範囲を絞ることでマージ処理コストを削減する．P-SORTを実装し性能評価した結果，従来のデータ並列方式では16コアで6.1倍の性能向上に留まるのに対し，P-SORTでは16コアで13倍となることを確認した．今後は実行時のデータ再割当てに対応するなど，方式の拡張を検討する．また，レイテンシの評価などさらに詳細な評価を進める予定である．

参　考　文　献

[1] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, “Scalable distributed stream processing”, Proc. of CIDR 2003.

[2] Y. Xing, S. Zdonik, and J. Hwang, “Dynamic load distribution in the Borealis stream processor”, Proc. of ICDE 2005.

[3] Y. Xing, J. Hwang, U. Cetintemel, and S. Zdonik, “Providing resiliency to load variations in distributed stream processing”, Proc. of VLDB 2006.

[4] T. Johnson, M. S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck, “Query-aware partitioning for monitoring massive network data streams”, Proc. of SIGMOD 2008.

[5] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, “Flux: an adaptive partitioning operator for continuous query systems”, Proc. of ICDE 2003.

[6] Aleri, “Coral8 Technology Overview”, http://www.aleri.com/

[7] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo, “SPADE: The System S Declarative Stream Processing Engine”, Proc. of SIGMOD 2008.

[8] StreamBase, “StreamBase 6.5.3 Documentation”, http://www.streambase.com/developers-home.htm

[9] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, “Query Processing, Resource Management, and Approximation in a Data Stream Management System”, Proc. of CIDR 2003.

[10] A. Arasu, S. Babu, and J. Widom, “The CQL continuous query language: semantic foundations and query execution”, The VLDB Journal, Vol. 15, 2006.

[11] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management”, The VLDB Journal, Vol. 12, 2003.

[12] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maie, “Out-of-order processing: a new architecture for high-performance stream systems”, Proc. of VLDB 2008.